Контрольная 2

Задача: "калькулятор"

- система классов Expression, Val, Add, Mul, Var
- eval() без контекста и с контекстом
- simplified()

Сложности

- управление памятью
- иерархия классов
- передача контекста
- реализация simplified()

Управление памятью

Очевидно, что объекты типов Add и Mul владеют переданными в них указателями. И они должны освобождать память в деструкторе.

Это можно сделать 2 способами:

- вручную
- `std::unique_ptr`

В пользу ручной работы - API, в котором все публичные функции принимают и возвращают голые указатели.

В пользу unique_ptr - ну, просто так удобнее.

Пример, как можно сделать с unique_ptr

```
using EPtr = std::unique_ptr<Expression>;

class Add: public Expression {
public:
    Add(Expression* lhs, Expression* rhs):
        Add(EPtr(lhs), EPtr(rhs)) {}
    // ~Add() = default;
private:
    Add(EPtr lhs, EPtr rhs) :
        lhs_(std::move(lhs)), rhs_(std::move(rhs)) {}
    EPtr lhs_;
    EPtr rhs_;

    .....
};
```

Почему два конструктора? Потому что передача unique_ptr в конструктор пригодится при реализации simplified().

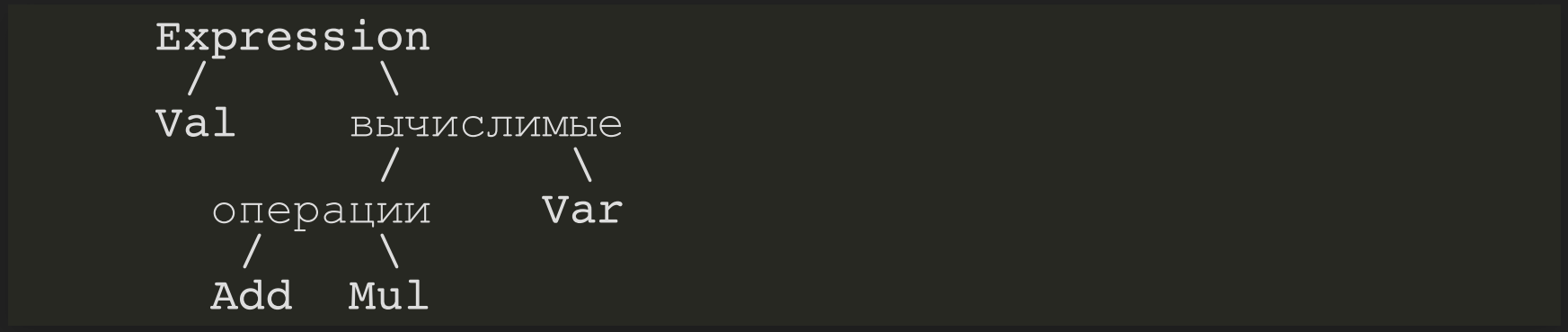
Без unique_ptr - очевидно

```
class Add: public Expression {
public:
    Add(Expression* lhs, Expression* rhs) : lhs_(lhs), r
    ~Add() {
        delete lhs_;
        delete rhs_;
    }
private:
    Expression* lhs_;
    Expression* rhs_;

    .....
};
```

Иерархия классов

Это не столько сложность, сколько вопрос: будем строить иерархию или копипастить.



У обеих операций по два операнда, поэтому можно попробовать какие-то общие части вынести в базовый класс, скажем, Binary.

Val безусловно вычисляется без контекста, тогда как остальные могут потребовать контекст (особенно, для упрощения). Нам потребуется отличать одно от других.

Пример выноса общих частей:

```
class Binary: public Expression {
// владение операндами
private:
    Expression* lhs_;
    Expression* rhs_;
    // ну или std::unique_ptr
protected:
    Binary(Expression* lhs, Expression* rhs) :
        lhs_(lhs), rhs_(rhs) {}
    ~Binary() override {
        delete lhs_;
        delete rhs_;
    }
```

```
// функции печати
protected:
    virtual bool parentheses() const = 0;
    virtual char symbol() const = 0;
public:
    void print(std::ostream& ost) const override {
        if (parentheses()) ost << "(";
        lhs_->print(ost);
        ost << " " << symbol() << " ";
        rhs_->print(ost);
        if (parentheses()) ost << ")";
    }
    . . . . .
```

```
    . . . . .
// функции вычисления
protected:
    virtual int64_t calc(int64_t lhs, int64_t rhs) const
public:
    int64_t eval() const override {
        return calc(lhs_->eval(), rhs_->eval());
    }
    int64_t eval(const Context& ctx) const override {
        return calc(lhs_->eval(ctx), rhs_->eval(ctx));
    }
    . . . . .
```

```
    . . . . .
// функции упрощения
protected:
    virtual Expression* construct(
        Expression* lhs, Expression* rhs) const = 0;
public:
    Expression* simplify(const Context& ctx) const overr
        . . . . .
        auto simplified_lhs = lhs_->simplify(ctx);
        auto simplified_lhs = rhs_->simplify(ctx);
        . . . . .
        calc(simplified_lhs->eval(), simplified_lhs->eva
        . . . . .
        construct(simplified_lhs, simplified_rhs)
        . . . . .
    }
    . . . . .
```

Отличать константные выражения от неконстатных можно сделать несколькими способами. Какими?

- на уровне типов: завести функцию `virtual bool IsValue() const`
- средствами RTTI C++: `dynamic_cast<Val*>`
- на уровне объектов: завести флажок `bool is_value_` у Expression
- пытаться сделать `eval()` и ловить исключение

Передача контекста

Это, казалось бы, простое действие, но не у одного человека возникла типичная ошибка.

```
virtual int64_t eval(Context ctx) const;
```

Да, контекст CopyConstructible, и его, технически, можно передавать по значению. Но цена вопроса довольно дорогая.

Чтобы избавить себя от подобных искушений, можно в некоторых случаях явно запрещать семантику копирования.

```
class Context {
public:
    Context() = defaut;
    Context(Context&&) = default;
    Context(const Context&) = delete;
    . . . . .
private:
    std::unordered_map<std::string, int64_t> values_;
};
```

Самая сложная часть: упрощение!

- Несколько человек неправильно поняли задание: вместо упрощения формул реализовали подстановку
- Небрежности с голыми указателями приводят к утечкам

В чём разница между подстановкой и упрощением

- подстановка: `(x + y * z)` при y=2, z=3 ⇒ `(x + 2 * 3)`
- упрощение: `(x + y * z)` при y=2, z=3 ⇒ `(x + 6)`
- оптимизация: `(x + 0)` ⇒ `x`, `x * 1` ⇒ `x`, `x * 0` ⇒ `0`

Нам нужно было только упрощение, без оптимизации.

Попробуем написать упрощение

Для Val - тривиально

```
Expression* Val::simplified(const Context& /*ctx*/) {
    // просто возвращаем копию себя
    return new Val(*this);
}
```

Для Var - чуть сложнее

```
Expression* Var::simplified(const Context& ctx) {
    // если можно вычислить как константу
    return new Val(eval(ctx));
    // иначе - возвращаем копию себя
    return new Var(*this);
}
```

Для двуместных операций - ещё сложнее

```
Expression* Add::simplified(const Context& ctx) const {
    // если данное выражение можно вычислить как констан
    return new Val(eval(ctx));
    // иначе - реконструировать из упрощённых подвыражен
    return new Add(
        lhs_->simplified(ctx),
        rhs_->simplified(ctx));
}
```

Как осуществлять ветвление "можно/нельзя вычислить"?

- У нас есть готовый механизм исключений, но он дорогой
- Можем расписать действия по шагам

Для Var - используем Context::varIsSet

```
Expression* Var::simplified(const Context& ctx) {
    if (ctx.varIsSet(name_)) {
        // тут уж точно не будет исключений
        return new Val(ctx.getVar(name_));
    }

    return new Var(*this);
}
```

Допустим, мы решили сделать на исключениях

```
Expression* Add::simplified(const Context& ctx) const {
    try {
        return new Val(eval(ctx));
    } catch (const std::runtime_error&) {
        return new Add(
            lhs_->simplified(ctx),
            rhs_->simplified(ctx));
    }
}
```

Или даже, по аналогии с переменными, ввели функцию

```
virtual bool canEval(const Context& ctx) const;
```

Как тут дела с вычислительной сложностью? Представим себе формулу вида

```
((((x + 1) + 2) + 3) + 4) + 5)
```

Сколько раз мы попытаемся вычислить подвыражения, если x отсутствует в контексте?

Можем разменять время на память

```
Expression* Add::simplified(const Context& ctx) const {
    Expression* lhs1 = lhs_->simplified(ctx);
    Expression* rhs1 = rhs_->simplified(ctx);
    // какое время жизни у этих объектов?

    // если данное выражение можно вычислить как констан
    if (lhs1->isVal() && rhs1->isVal()) {
        // ничего не забыли?
        return new Val(lhs1->eval() + rhs1->eval());
    }

    // ничего не случится?
    return new Add(lhs1, rhs1);
}
```

Вот тут удобно использовать std::unique_ptr, чтобы точно ничего не забыть.

```
Expression* Add::simplified(const Context& ctx) const {
    std::unique_ptr<Expression> lhs1(lhs_->simplified(ct
    std::unique_ptr<Expression> rhs1(lhs_->simplified(ct
    // если из rhs_->simplified(ctx) вылетит исключение,

    if (lhs1->isVal() && rhs1->isVal()) {
        Expression* result = new Val(lhs1->eval() + rhs1
        // если из new Val вылетит исключение, очистим l
        lhs1.reset();
        rhs1.reset();
        return result;
    }

    return new Add(std::move(lhs1), std::move(rhs1));
    // если из new Add вылетит исключение, очистим време
}
```